5. Stacks

Motivation

Stack Operations

Stack Interface

Checking for a^nb^n

Implementing a^nb^n

Checking for Braces

Checking for Palindromes

Array Based Stacks

Array Implementation

Pointer Based Stacks

Pointer Implementation

Postfix Expressions

Stack Based Flood Fill

Stack Discussion

Motivation

- Stacks were invented as an abstract data type (ADT) for “last in first out” storage.

- Think of a pile of dishes in your cupboard.
  - We put clean dishes away one at a time on top of a pile.
  - When we want to use a dish we take the top dish.
  - Thus we use dishes in a “last in first out” way.

- Many programming problems can be solved by using a stack to store data.

Stack Operations

The Stack ADT usually has the following operations:

- create - Makes a new empty stack.
- destroy - Deletes all data on the stack.

- push - Stores data on top of all other data in stack.
- pop - Retrieves the data item on top of the stack.
- top - Retrieves the data on top, but does not remove it. (Same as pop/push)

- full - Checks if stack has room for more data.
- empty - Checks if stack has any data available.

Stack Interface

The following C++ class allows users to store characters on a stack:

```
class Stack{
public:
    Stack();
    ~Stack();

    void push(char item);
    char pop();
    char top();
    bool isFull();
    bool isEmpty();

private:
    TBA
};
```

Note:

You can easily change storage to another data type by changing the **char** to another type.

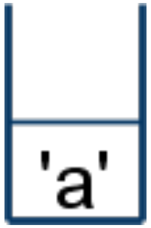
Checking for a^nb^n

- Assume you are given an unknown number of characters from the user.
- How can you check to see if they have entered something of the from $\mathbf{a^nb^n}$ where **n >=0**.
- One solution is to push 'a's on the stack as you read them, and pop 'a's when you read a 'b'.

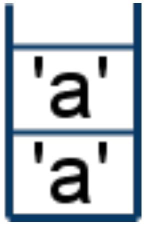
Checking for a^nb^n

Example: user enters "aabb".

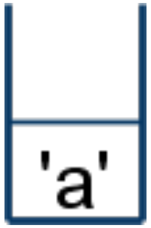
1. read 'a', push



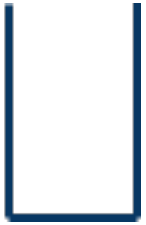
2. read 'a', push



3. read 'b', pop



4. read 'b', pop



*Stack is empty so input was valid.

Implementing a^nb^n

```
bool check_anbn( char str[]){
    Stack s;    char ch;

    // process input
    for(int i=0; i < strlen(str); i++){
        if(str[i] == 'a')
            s.push('a');
        else if(str[i] == 'b')
            ch = s.pop();
    }

    // check if stack is empty
    if(s.empty())
        return true;
    else
        return false;
}
```

- This function has a number of logic errors.
- Can you find them?

Implementing a^nb^n

```
bool check_anbn( char str[] ){
    Stack s;    char ch;

    // process a's
    int i=0;
    while(((i<strlen(str))&&(str[i]=='a'))&&(!s.full())){
        s.push('a');
        i++;
    }

     // process b's
    while((i<strlen(str))&&(str[i]=='b')&&(!s.empty())){
        ch=s.pop();
        i++;
    }

    // check for success
    if(s.empty() && ( i==strlen(str)))
        return true;
    else
        return false;
}
```

Checking for Braces

- Assume you are given a C++ program where all comments have been removed.
- How can you check if the braces { } are balanced?
- You could just count '{' and '}' but this would not check ordering.
- One solution is to use a stack and push '{' when you read it and pop when you read '}'.
- Braces are balanced if stack is empty at the end (and we didn't have a stack underflow while checking).

Checking for Braces

```
bool check_braces()
{
    Stack s;    char ch;

    //loop reading characters
    while( cin >> ch){
        if( ch=='{' )
            s.push(ch);
        else if( ch=='}' ){
            if( !s.isEmpty() )
                ch = s.pop();
            else
                return false;
        }
    }
    return ( s.empty() ); //check for success
}
```

- Do you think this code will work for all programs?
- What about this program?

Checking for Palindromes

- Assume you are given a string of known length.
- How can you check to see if it is a palindrome (the same read forwards or backwards)?
- One solution is to use a stack and push half of the string on, and check if characters match when processing the second half of the string and popping the stack.
- This approach only works if know how long the string is in advanced or the middle is marked by some special character. (We will see a solution that works for any input later.

Checking for Palindromes

```
bool check_palindrome( char str[]){
    Stack s;    char ch;

    //push first half
    int i=0;
    while( i<strlen(str)/2 ){
        s.push(str[i]);
        i++;
    }

    //pop second half
    while( i<strlen(str) ){
        ch=s.pop();
        if( ch!=str[i] )
            return false;
        i++;
     }
     return true;
}
```

Checking for Palindromes

- Assume input was "abccba", length=6

i	Stack
0	a
1	ab
2	abc
3	ab
4	a
5	

- Program returns true at the end.

- Assume input was "xyzyx", length=5

i	Stack
0	x
1	xy
2	x

- Program returns false since y!=z

- Need to correct program logic to handle odd length input strings. (we must skip the middle character)

- Add "if(strlen(str)%2==1) i++;" to code.

Array Based Stacks

- We can implement a stack using a fixed size array and an integer "top" that keeps track of the index of the top item.

Eg:

3	1	4	1	5	
0	1	2	3	4	

top=4

3	1	4	1	5	9	
0	1	2	3	4	5	

top=5 after push(9)

3	1	4	1	5	
0	1	2	3	4	

top=4 after pop

3	1	4	1	
0	1	2	3	

top=3 after pop

- We need to handle potential stack overflow and underflow. (pop when top<0)

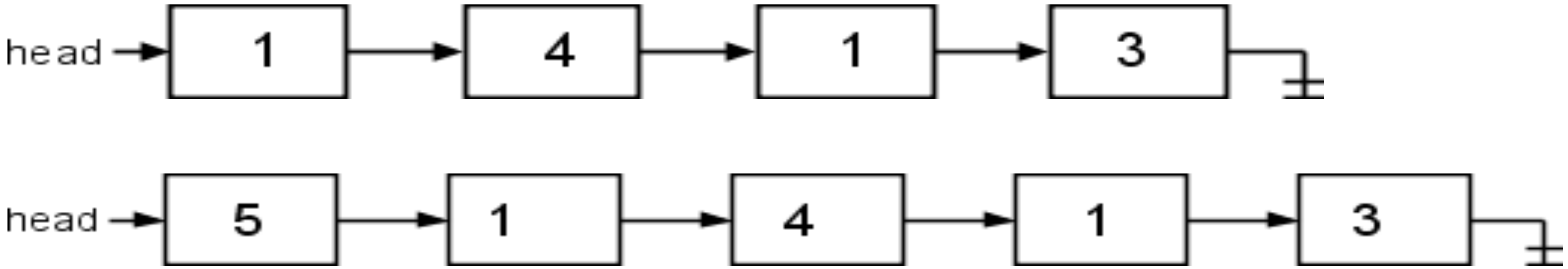
Array Implementation

```
void push( char item )
{
    //check size
    if( top<MAX_SIZE )
    {
        top++;
        data[top]=item;
    }

    // print error message
    else
       cout<<"stack overflow\n";
}
```

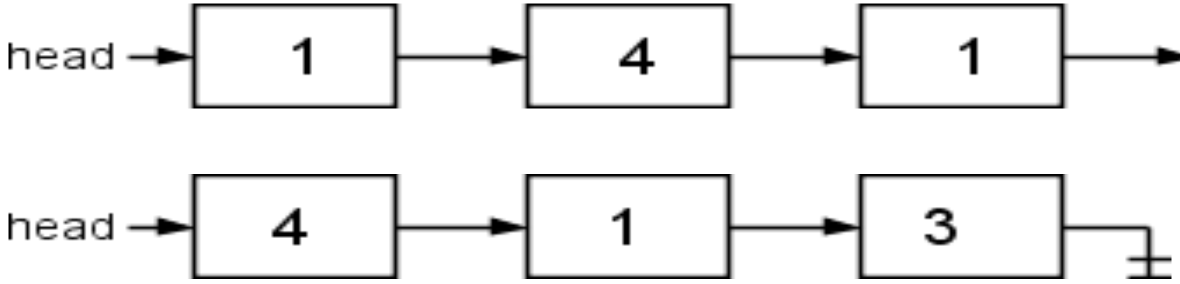
```
char pop()
{
    //check size
    if( top >=0 )
        return data[top--];

    // handle stack underflow
    else
        return '\0';
}
```

Pointer Based Stacks

- We can create a dynamic stack using a subset of linked-list operations, inserting and removing at the head.



Stack after push(5)



Stack after two pops

- Stack can never get full unless we run out of memory.

Pointer Implementation

```
void push( char item )
{
    stack_node* tmp;
    tmp=new stack_node();
    tmp->data=item;
    tmp->next=head;
    head=tmp;
}
```

```
char pop()
{
    //check empty stack
    if( head!=null )
    {
        stack_node *tmp=head;
        head=tmp->next;
        char item=tmp->data;
        delete tmp;
        return item;
    }

    // handle stack underflow
    else
        return '\0';
}
```

Pointer Implementation

```
void push( char item ){
    stack_list.insert_head(item);
}

char pop(){

    //check empty stack
    if( !stack_list.empty() ){
        char item=stack_list.remove_head();
        return item;
    }

    // handle stack underflow
    else
        return '\0';
}
```

Postfix Expressions

- A postfix expression is written with operators following the values.

Eg: "2 3 +" really means "2 + 3"
"2 3 + 5 *" means "(2 + 3) * 5"

- It is easy to evaluate a postfix expression using a stack to store values.
- When we see an integer, we push it on the stack.
- When we see an operator, we pop the top two values, perform the operation, and push the result on the stack.
- The value on the stack at the end is the final result.

Postfix Expressions

```
int postfix(){
    int_stack s;    char str[str_size];

    //loop untill end of file
    while( cin >> str ){

        if(str[0]=='+')    //handle addition
            s.push( s.pop() + s.pop() );

        if(str[0]=='*')    //handle multiplication
            s.push( s.pop() * s.pop() );

        else               //handle numbers
            s.push( atoi(str) );
    }
    return s.top();    //return answer
}
```

Postfix Expressions

- Assume user enters a sequence of numbers and operators separated by spaces.

input	Stack
2	2
3	2 3
+	5
5	5 5
*	25
eof	

- push 2
- push 3
- push 2+3=5
- push 5
- push 5*5=25

- What happens if user enters "4 5 + 6"?
- What happens if "7 + 8" is entered?

Stack Based Flood Fill

- The floodfill function can be implemented on a stack instead of using recursion.

- When we visit an unmarked pixel we push the coordinates of 4 neighbors on to the stack.
- To decide where to go next, we pop the top coordinate from the stack and go there.

- We will end up visiting all the pixels in the polygon in the same order as the recursive version.

Stack Based Flood Fill

```
void floodfill( int x, int y, int color) {
    int_stack s;
    s.push(x);    s.push(y);    //push first point on stack

    //pop and process points on stack
    while(!s.empty()) {
         //get point
        y=s.pop();    x=s.pop();

        if( pixel[y][x] != color ) {      //fill point and neighbors
            pixel[y][x]=color;
            s.push(x);    s.push(y-1);
            s.push(x);    s.push(y+1);
            s.push(x-1);    s.push(y);
            s.push(x+1);    s.push(y);
        }
    }
}
```

Stack Based Flood Fill

- Assume we are given the following polygon to fill.

3	x	x	x	x
2	x			x
1	x			x
0	x	x	x	x
	0	1	3	4

- Seed point is (1,1) for call to flood-fill.

Action	Stack
	(1,1)
fill(1,1)	(1.0)(1,2)(0,1)(2,1)
fill(2,1)	(1,0)(1,2)(1,1)(2,0)(2,2)(1,1)(3,1)
	(1,0)(1,2)(1,1)(2,0)(2,2)(1,1)
	(1,0)(1,2)(1,1)(2,0)(2,2)
fill(2,2)	(1,0)(1,2)(1,1)(2,0)(2,1)(2,3)(1,2)(3,2)
	(1,0)(1,2)(1,1)(2,0)(2,1)(2,3)(1,2)
fill(1,2)	(1,0)(1,2)(1,1)(2,0)(2,1)(2,3)(1,1)(1,3)(0,2)(2,2)
	(1,0)(1,2)(1,1)(2,0)(2,1)(2,3)(1,1)(1,3)(0,2)
	etc.

Stack Based Flood Fill

- Notice that floodfill's push's were done in the opposite order from the recursive calls in our other floodfill function.
- Since stacks are "last in first out" the last push was popped off and filled first.
- Hence the pixels were filled in the same order as in the recursive version.
- We can cut down on the amount of data on the stack by checking to see if the pixel has been filled before pushing it.

```
if(pixel[y-1][x]!=color)
{   s.push(x);    s.push(y-1);    }
```

Stack Discussion

- Stacks are a simple ADT to implement using arrays or linked lists.

- Stacks are useful "memory" for problems that require symmetry.

- Stacks are also useful for simulating recursive algorithms -- keeping track of work to be done later.